

me

0420 06-11-01 #14

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
Group Art Unit Not yet assigned

In re

Patent Application of

Keith Rautenback, et. al.

Serial No. 09/858,426

Filed: May 16, 2001

Examiner: Not yet assigned

"METHOD AND STRUCTURE FOR REDUCING
SEARCH TIMES"



I, Karen J. Jurkowski, hereby certify that this correspondence is being deposited with the US Postal Service as first class mail in an envelope addressed to Assistant Commissioner for Patents, Washington, D.C. 20231, on the date of my signature.


Signature

6/18/01
Date of Signature

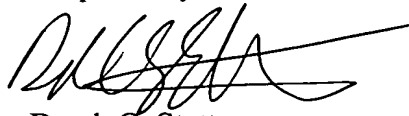
SUBMISSION OF PRIORITY DOCUMENT

Assistant Commissioner for Patents
Washington, DC 20231

Sir:

Enclosed is a priority document for GB 9825102.8 for filing in the above-identified application.

Respectfully submitted,



Derek C. Stettner
Reg. No. 37,945

Docket No.: 048487-9050-00
Michael Best & Friedrich LLP
100 East Wisconsin Avenue
Milwaukee, Wisconsin 53202-4108

(414) 271-6560

T:\CLIENTA\048487\9050\A0192581



THIS PAGE BLANK (USPTO)



INVESTOR IN PEOPLE



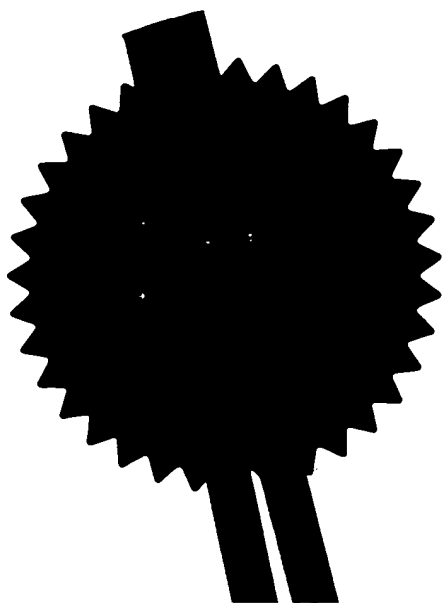
The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

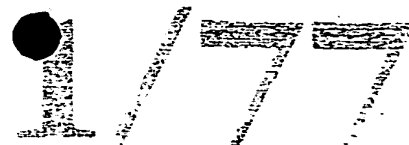


Signed

Dated 30 May 2001

**CERTIFIED COPY OF
PRIORITY DOCUMENT**

THIS PAGE BLANK (USPTO)



17NOV98 E405162-3 D02000
P01/7700 0.00 - 9825102.8

The Patent Office

Cardiff Road
Newport
Gwent NP9 1RH

Request for grant of a patent

(See the notes on the back of this form. You can also get an explanatory leaflet from the Patent Office to help you fill in this form)

1. Your reference

PDC\20996

2. Patent application number

(The Patent Office will fill in this part)

16 NOV 1998

9825102.8

3. Full name, address and postcode of the or of each applicant (underline all surnames)

Patents ADP number (if you know it)

If the applicant is a corporate body, give the country/state of its incorporation

Insignia Solutions PLC
The Mercury Centre
Wycombe Lane, Wooburn Green
High Wycombe
Buckinghamshire HP10 0HH
United Kingdom

69 34616002

4. Title of the invention

Computer System

5. Name of your agent (if you have one)

"Address for service" in the United Kingdom to which all correspondence should be sent (including the postcode)

MATHYS & SQUIRE
100 Grays Inn Road
London WC1X 8AL

Patents ADP number (if you know it)

1081001

6. If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and (if you know it) the or each application number

Country

Priority application number
(if you know it)

Date of filing
(day / month / year)

7. If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application

Number of earlier application

Date of filing
(day / month / year)

8. Is a statement of inventorship and of right to grant of a patent required in support of this request? (Answer 'Yes' if:

- a) any applicant named in part 3 is not an inventor, or
 - b) there is an inventor who is not named as an applicant, or
 - c) any named applicant is a corporate body.
- See note (d))

YES

Patents Form 1/77

9. Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document

Continuation sheets of this form

Description 63

Claim(s) -

Abstract -

Drawing(s) 6 + 6



10. If you are also filing any of the following, state how many against each item.

Priority documents

Translations of priority documents

Statement of inventorship and right to grant of a patent (*Patents Form 7/77*)

Request for preliminary examination and search (*Patents Form 9/77*)

Request for substantive examination (*Patents Form 10/77*)

Any other documents
(please specify)

11. I/We request the grant of a patent on the basis of this application.

Signature



Date

16 November 1998

12. Name and daytime telephone number of person to contact in the United Kingdom

Dr Paul Cozens

0171 830 0000

Warning

After an application for a patent has been filed, the Comptroller of the Patent Office will consider whether publication or communication of the invention should be prohibited or restricted under Section 22 of the Patents Act 1977. You will be informed if it is necessary to prohibit or restrict your invention in this way. Furthermore, if you live in the United Kingdom, Section 23 of the Patents Act 1977 stops you from applying for a patent abroad without first getting written permission from the Patent Office unless an application has been filed at least 6 weeks beforehand in the United Kingdom for a patent for the same invention and either no direction prohibiting publication or communication has been given, or any such direction has been revoked.

Notes

- If you need help to fill in this form or you have any questions, please contact the Patent Office on 0645 500505.*
- Write your answers in capital letters using black ink or you may type them.*
- If there is not enough space for all the relevant details on any part of this form, please continue on a separate sheet of paper and write "see continuation sheet" in the relevant part(s). Any continuation sheet should be attached to this form.*
- If you have answered 'Yes' Patents Form 7/77 will need to be filed.*
- Once you have filled in the form you must remember to sign and date it.*
- For details of the fee and ways to pay please contact the Patent Office.*

COMPUTER SYSTEM

This invention relates, in its most general aspects, to a computer system and to improvements in the performance of various operations within such a system.

5

In recent years, there have been developments in programming languages towards what is known as an Object-Oriented Language. In these developments, concepts are regarded as "objects", each carrying with it a set of data, or attributes, pertinent to that object, as well as information relating to so-called "methods", ie sub-routines, that can be performed on that object and its data. This is well known to those skilled in the art of computing and/or programming.

The advent and rapid advancement in the spread and availability of computers has led to the independent development of different types of system, such as the IBM and IBM-compatible PC running IBM-DOS or MS-DOS or MS-Windows applications, the Apple Macintosh machines running their own Apple System operating system, or various Unix machines running their own Unix operating systems. This proliferation of independent systems has led to useful applications being available only in one format and not being capable of running on a machine for which the application was not designed.

Under such circumstances, programmers have devised software which "emulates" the host computer's operating system so that a "foreign" application can be made to run successfully in such a way that, as far as the user is concerned, the emulation is invisible. In other words, the user can perform all of the normal functions of say a Windows-based application on a Unix machine using a Unix-based operating system without noticing that he is doing so.

A particularly notable product of this type is that developed by Insignia Solutions of High Wycombe, GB and Santa Clara, California, USA and known under the name "SoftWindows 2.0 for Powermac". This software enables a physical Macintosh computer to emulate a PC having an Intel 80486DX processor and 80487 maths co-processor plus memory, two hard disks, IBM-style keyboard, colour display and other features normally found on younger versions of the PC-type of computer.

Furthermore, there is an ever-increasing demand by the consumer for electronics gadgetry, communications and control systems which, like computers, have developed independently of one another and have led to incompatibility between operating systems and protocols. For example, remote-control devices for video players, tape
5 players and CD players have similar functions, analogous to "play", "forward", "reverse", "pause" etc, but the codes for transmission between the remote control, or commander, operated by the user may not be compatible either between different types of equipment made by the same manufacturer or between the same types of equipment made by different manufacturers. There would be clear benefits of having
10 software within the equipment which can produce for example the correct "play" code based upon a "play" command regardless of the specific hardware used in the equipment. Such software is commonly known as a "Virtual Machine".

Other uses and applications are legion: for example, set-top boxes for decoding
15 television transmissions, remote diagnostic equipment, in-car navigation systems and so-called "Personal Digital Assistants". Mobile telephones, for instance, can have a system upgrade downloaded to them from any service provider.

Emulation software packages tend to have certain features in common, notably that
20 they are not general purpose but are dedicated. They are of most benefit in rapid development areas and have a distinct advantage in enabling manufacturers to cut costs. In particular, they can divorce software from the physical machine ie the effect of the software in the physical machine can be altered by the emulating software without having to go into the machine's native software to implement those changes.

25 The specific object-oriented language used in some of the implementations described later is that known as Java (registered trade mark to Sun Microsystems Corporation). Some of the following implementations will enable Java to be used in smaller devices than is currently possible because of the improved performance and/or reduced
30 memory footprint. Future uses projected for embedded software (virtual machines) include computers worn on the body, office equipment, household appliances, and intelligent houses and cars.

While it is recognised that there are clear advantages in the use of virtual machines,

especially those using object-oriented languages, there are naturally areas where it is important and/or beneficial for some of the operations that are carried out within the system to be optimised. These may include reducing the memory requirement, increasing the speed of operation, and improving the "transparency" of the system when embedded in another system.

There now follows description of various improvements in the operation of computer systems in accordance with the teaching of the present invention. They fall generally into the following headings:

10

- Dynamic compilation of the dominant path
- Pre-exception condition checks
- Outliers for spatial separation of infrequent code etc
- Dispatch mechanism for interface methods
- 15 • Return barriers: Minimising blocking while a thread's stack is being inspected by a concurrent garbage collector
- Stack walking to allow efficient compiled code deletion in the multi-threaded environment of a unified stack virtual machine
- Grey packets - low contention grey object sets for concurrent marking garbage collecting in a highly multi-threaded environment
- 20 • (Executing) Device driver interrupt handlers written in Java
- Use of class loader to allow direct invocation of non-final instance methods
- Reducing search times for unordered lists in multi-threaded environment
- Method for automatic test and verification of dynamically compiled code in a virtual machine
- 25

Reference will be made, where appropriate, purely by way of example, to the accompanying figures of the drawings (which represent schematically the above improvements) in which:

30

Figure 10 shows paths of execution

Figure 12 shows the comparative costs of compiling dominant paths

Figure 20 shows the memory arrangement in a computer system

Figure 40 illustrates the hierarchical structure in object-oriented programming

Figure 41 shows the arrangement of data stored in dispatch tables

Figure 42 shows the application of an interface hash table to a dispatch table

Figure 50 shows the use of return barriers in activation frames of a stack

Figure 70 shows the application of grey packets in a concurrent garbage

5 collection system

Figures 100, 101 and 102 show a loop arrangement as applied to unordered lists in a multi-threaded environment.

Figure 200 shows certain components of the virtual machine

10

Dynamic Compilation of the Dominant Path:

Technical field

- 5 This invention is preferably related to the optimisation of the runtime representation of object-oriented computer languages by means of runtime compilation technology and preferably to the optimisation of the runtime representation of object-oriented computer languages by means of runtime compilation technology. Aspects of the invention are related to optimised execution of virtual machines, and in particular
- 10 Java virtual machines.

Background information

trace scheduling

optimising compilers

dynamic compilation

- 15 profile guided optimisations

just in time compilers

Java VM spec

Problems with known techniques

- 20 Generally, compilers of the runtime representation of computer languages and in particular so-called Just-in-time (JIT) compilers, compile the representation of a whole method at a time, or a larger unit (for example, a file or one of many classes at a time). Often a significant portion of an application relates to handling exceptional situations, or rarely executed code. Typically, the compiler blocks any further progress of the application until the compilation completes.
- 25 The conventional compilation approach therefore spends time compiling code which is rarely executed, and the compiled result occupies space which would have not been needed if the rarely executed code were not present. Optimisation

opportunities are often reduced by having to cater for control paths through the rarely executed code.

Offline compilers which use profile input from a previous run of the application can often optimise the frequently executed paths of an application to mitigate the latter
5 problem. However they still must compile every path through the application, and cannot easily react when an application exhibits different behaviour, to that of the profile run.

Solution to the problems

- The Invention described in this application involves any, some or all of the
10 following features, in any combination:
1. Compile fragments of code for the dominant path rather than whole methods.
 2. Use execution history to determine which paths through the application are the dominant ones.
 3. Use a fallback interpreter to interpret infrequently executed code.
 - 15 4. Have an online compilation system which can compile code on demand as the application executes. This system does not block progress of the application. The system runs as a separate thread, whose priority is adaptive.
 5. Have the ability to incorporate new fragments of code into a running multi-threaded system.
 - 20 6. Support removal of fragments of code from a running multi-threaded system.
 7. Constrain the amount of memory used by the dynamic compiler during its execution at any time.

The invention described in this application aims to:

- 25 - Reduce the performance impact of online compilation.
- Generate code which is optimised for the dominant paths through an application.
- Allow better optimisation of code, within time and memory constraints.
- Reduce the storage overhead of compiled code which is rarely executed.
- Improve application responsiveness in a multi-threaded computer system.
30 - Reduce the amount of memory used by the compiler itself.

Accordingly, the invention provides a computer system containing a compiler for

compiling the operating code of an application, in which only dominant path (or near dominant path) fragments of the code are compiled.

5 This technique can afford the primary advantage of enhancing performance and reducing compiled space. It is important for a small memory application and involves a mixture of trade offs between memory size, compilation time and performance.

10 In its preferred form, it also enables the use of key optimisation techniques, involving loops and inlining, without the overhead of global dataflow analysis, and hence allows the compiler itself to execute much faster than compilers that do perform global dataflow analysis. The memory usage of the compiler itself is also much lower.

15 In the system as defined, advantageously only the dominant path of execution is compiled, rather than all the paths through the code, while the remaining paths are interpreted.

20 It is a particularly preferred feature that the compiler is operating on-line, in the sense that as the operating code is running parts of it are being compiled; what is termed the dominant path may be constantly changing as execution of the code progresses.

Also, according to the invention, there is provided a method of operating a computer system containing a compiler for compiling the operating code of an application, the method comprising compiling only the dominant path fragments of the code.

25 The method can enhance the performance and reduce the compiled space requirement of the computer system and the memory space requirements of the compiler itself.

30 Advantageously, information identifying the dominant path is provided from the execution history of the code. The execution history information is preferably derived dynamically as the program runs. The execution history information is advantageously captured from a previous run of the code.

In its preferred embodiment, infrequently executed code is interpreted in a fallback interpreter, whereby preferably execution of the code can continue without the need

for compiled code for the infrequently executed code.

Advantageously, an online compilation system is provided which can compile code on demand as the application/program executes whereby compilation information can
5 be generated in response to the appearance of a new frequently executed path.

When the computer system is operating in a multi-threaded system, new fragments of code are preferably incorporated into the multi-threaded system, whereby preferably to achieve smoother operation without stopping running threads.

10

Detailed description of the invention

1. Compile fragments of code for the dominant path rather than whole methods.

The compiler takes as input the runtime representation of the source program, and execution history information. The execution history information could be live (i.e.
15 dynamically changing as the program runs), or captured from a previous run of the program.

Execution history information is combined with structural information determined from the runtime representation of the program source, to establish what is the dominant path of the program the compiler should compile. Unexecuted code is
20 preferably never included in the dominant path.

The compiler treats the dominant path as a super-block fragment, laying the code out sequentially, even though the program source may not be. Branches and tests are adjusted where necessary to make the dominant path fall-through. Code and registers are optimised with the assumption that the dominant path will be followed to the end.
25 This improves performance on modern processor architectures. Critically, the dominant path only exposes one external entry point. This greatly simplifies and enhances optimisations.

As shown in Figure 10, where the path of execution would leave the dominant path, the appropriate run-time tests are inserted with a forward branch 100 to some stub

code referred to as an "Outlier". 102. The outlier stub updates any state which the dominant path has not written back yet, before transferring control out of the fragment. The mainline code of dominant paths are generally kept together, as are the outlier stubs as shown at 102. This improves performance on modern processors, especially where branch prediction software/hardware initially assumes that forward branches are less likely. It also provides better instruction cache behaviour.

Compiling dominant paths of execution allows loop optimisations and inlining to be performed, while simplifying the analysis required for many optimisations. It obviates the need for the compiler to have to resolve symbolic references.

- 10 Often exceptions need to be recognised in the middle of a loop after some global state has changed. The exception check can be performed early outside the loop, forcing the code into the fallback interpreter, thus allowing the check to be removed from the loop, and code motion to be performed in the presence of those exceptions.

The fallback interpreter will execute the loop and recognise the exception at the right time, albeit more slowly. It is assumed that exceptions rarely occur, and therefore the benefits of the optimised loop will outweigh the disadvantages.

When the compiler is being executed online as the application is executed, the compilation overheads are often critical. By only compiling the dominant path, the compiler is simpler, quicker, and uses less memory for its analysis and therefore can afford to perform more optimisations than would otherwise be feasible, especially in a small memory system.

2. Use execution history to determine which paths through the application are the dominant ones.

Execution history is captured as the application executes. It is maintained at the block level, when a transfer of control occurs.

For each block an entry count and list of successors is kept with a count associated with each. These counts act as an indicator of popularity. Execution history records also contain an indication of what instruction caused the transfer of control which

ends the block, and state information which is updated as a result of compilations. Only blocks that have executed up to the transfer of control are candidates.

When memory is constrained, execution history records are recycled in two ways. Firstly, the list of successors is limited to a small number, and when a new successor is encountered the least popular existing successor is replaced with the new one. When there are no free execution history records, all of the history records associated with the least frequently used method are moved to the free list.

Compilation of a fragment is triggered by the entry count of a block exceeding a given threshold. The threshold may be fixed, or dynamically tuned. However, if the state of the history block indicates that the block is already queued to be compiled, or is not compilable, it is ignored.

When a compilation triggers, the dominant path can be determined by following the most popular successors a block at a time, including following method calls.

Generally speaking, execution history of the running application is a good indicator of which paths are the dominant ones.

Execution history does not need to be accurate, and can be updated in a number of ways. Rather than track execution history in compiled code, which would slow execution down significantly, execution history is maintained by the fallback interpreter.

20 3. Have a fallback interpreter which interprets infrequently executed code.

Having a fallback interpreter means that when infrequent or exceptional code is executed, execution can continue without the presence of compiled code for it. The fallback interpreter maintains execution history. It also means that all issues to do with class resolution can be solely handled by the fallback interpreter.

25 4. Have an online compilation system which can compile code on demand as the application executes.

As and when application behaviour changes, a dynamic compiler can generate optimised code for any new frequently executed paths which show up. By running as a separate thread, this allows the application to continue useful work via the fallback

interpreter.

5. Have the ability to incorporate new fragments of code into a running multi-threaded system.

5 Smoother operation is obtained if a new fragment of code can be incorporated without stopping running threads.

6. Support removal of fragments of code from a running multi-threaded system.

Removal of code fragments is the key to being able to operate in restricted memory environments. It also allows code which was optimised for one dominant path to be replaced with different code when new dominant paths appear. Code can be compiled
10 with optimistic optimisations on the basis that they can be deleted if the optimistic assumptions under which the code was compiled are broken.

The fact that compilation costs can be radically reduced is illustrated by the schematic diagram in Figure 12 in which the comparative time taken up in profiling, compiling
15 and executing at full speed for the invention 120 and the typical prior art 122 are shown as a proportion of a 10-second time slot.

Use of the dominant path also allows the dynamic compiler to be memory constrained by truncating a fragment some way along the path when the compiler reaches its
20 budgeted memory limit. This is impossible in prior art compilers.

It is crucial in small memory computer systems that the compiler adheres to a memory budget. Prior art compilers typically view memory as an unlimited resource. Hence they may consume large amounts of memory during compilation, to build
25 internal representations of its input program, and to hold results of dataflow analysis etc.

In contrast, the dynamic compiler works within external configurable constraints imposed upon it at system start up or build time. It then compiles as much of a
30 fragment as it can within these constraints. If necessary, it truncates the fragment, by relying on the feedback interpreter to receive control at the truncation point. This is impossible in prior art compilers, where the unit of compilation is a method or

greater, and where no interaction with a fallback interpreter is available.

Pre-exception condition checks

Technical field

5

Object-oriented programs (preferably Java).

Background information

10 Java is a language rich in exceptions. Java state must be written to as dictated by the semantics of the Java program. Thus, in the following simple example, one cannot update "x" before the array access is executed, in case the array access "arr[i]" raises an "index out of bounds" exception. If the write to "x" was incorrectly moved before the access, and an exception did occur, we would now have an incorrect value
15 for "x".

```

        x = ....;
        b = 10
        for (int l=a; i<b; l++) {
            arr[i]++;
20         x = b
        }

```

Standard code-motion optimisations; such as loop invariance, are thus blocked in the presence of such exceptions, which act as barriers across which code cannot be
25 moved.

Problems with known techniques

In the above example, "x" is being written with a loop-invariant value (10). In the presence of the potential exception, we cannot move the write outside of the loop. If
30 "a" did not fall within the range of allowable index values for the array "arr", then the first access to "arr[i]" would raise an exception and "x" would have the same value extant at entry to the loop, and not the value 10. Moreover, the exception check itself executes within the loop body, hence incurring its own execution penalty.

Secondary Example

Another example could be where a function J involves division of a function or value I by another function $Q=n$. There may be certain circumstances where Q becomes
5 zero, leading to division by zero, a non-calculable function, such as follows:

```
Q=n
For (I=1; I<10; I=I+1)
  if (I= <9)
10      do something (J)
      J=I/Q;
```

We know from the initial value of Q that it will reach zero at some point but it is not
advisable to throw the exception too early for fear that the programme loop may have
15 executed something which is of value. It is not impossible for a loop to be circulated
a large number of times (perhaps on average 10 times) when executing an algorithm
such as illustrated.

Solution to the problem

20 The invention aims to allow optimisations relating to code motion in the presence of
exception conditions within loops, which in turn improves the execution speed of the
resulting compiled fragment.

25 The solution is achieved by use of "pre-exception condition checks", whereby the
compiled fragment contains equivalent checks placed prior to the loop entry point.

Advantageously, such a check critically relies upon the presence of the fallback
interpreter. If the check detects an exception condition, then control reverts to the
30 fallback interpreter without the possibility of re-entering the fragment at this loop
entry point. The fallback interpreter continues execution at the loop entry point, and
hence executes up to the point where the exception is encountered at its correct
control point, thus raising the exception with all Java state containing the correct
values. If the pre-exception condition check passes however, then the fragment is

safely usable, and any code motion optimisations are valid.

In the above example therefore, one could have moved the loop-invariant assignment of "x" out of the loop, so long as it follows the check. This allows omission of the
5 original exception check in the loop, which also offers improved performance.

A suite of such pre-exception checks are used, including:

- * early checkcast check
- 10 * early bounds check against the possible range of array index values
- * early null-reference check
- * early divide by zero
- * early object type check, to enable code motion and other early checks to be applied to inlined methods

15

The advantage of the invention is the choice of the fast route through the compiled fragment or the slow route through the fallback interpreter. The invention enables the fast route to take advantage of code motion (including exception condition checks) outside of a loop, even in the presence of exception conditions within the loop. This
20 choice is unavailable to prior compilers which have compiled the entire method and whose compiled methods do not have the ability to interact with an interpreter to field exception conditions.

By virtue of the invention, the performance of the compile fragment may be greatly
25 improved due to the ability to move code out of loops. Hence greater freedom is available to the dynamic compiler in its choice and application of optimisations which are not normally available to prior compilers.

According to alternative aspects of the invention, there is provided a computer system
30 comprising (preferably during the running of a program) means for compiling an exception check to identify the occurrence of an exception condition, and means for executing an exception, when identified by said exception check, in an interpreted language.

Optionally there may also be provided means for carrying out an exception check to identify the occurrence of an exception condition.

5 In another aspect, the invention provides a method of operating a computer system comprising the steps of: running a program; compiling an exception check to identify the occurrence of an exception condition, and executing an exception, when identified by said exception check, in an interpreted language.

10 Preferably, the exception check is carried out outside a processing loop, whereby preferably to avoid the need for the exception check to be carried out at each circulation of the loop. An advantage of the invention is the choice of taking the fast route through the compiler or the slow route through the interpreter which is not available to prior compilers which work off-loop.

15 It may be possible, according to the invention, to decide outside the loop that the exception will be reached at some future point in time. When that occurs, the exception is passed off to the interpreter and therefore there is no necessity for the loop to be checked in each circulation of the loop.

20 The exception check itself is compiled but interpretation of the exception itself outside the loop in the slower interpreter serves to save compilation time and reduces memory requirements but does not prejudice optimisation. In Java, exception handling is carried out at programming level.

Outliers for spatial separation of infrequent code etc

Technical field

- 5 Management of cache memory in a computer system.

Background information

- 10 In a computer system there are various levels of cache memory. It is clearly of benefit to the system, in terms of improved efficiency and therefore speed, if the caches themselves can be operated efficiently. In order to improve cache locality, one of the aims of the invention, it is beneficial to have in the fastest of the caches the compiled code that the dynamic compiler has produced.

15 Problems with known techniques

Prior art solutions do not enable efficient operation of cache memory.

Solution to the problem

- 20 Accordingly, the invention provides a computer system comprising means for storing substantially all of (and preferably only) the dominant path compiled code together in one memory region or cache, whilst, preferably, any other code is only stored in spatially separate regions. Such a memory layout typically maximises the amount of
25 useful code loaded into the cache.

- The invention also provides a method of operating a computer system comprising the steps of: compiling all of the dominant path code; and storing substantially all of said compiled code in one memory region, while preferably storing infrequently accessed
30 code in a separate region.

The infrequently accessed code is termed an "outlier" since it lies out of the normal memory region for executed code. In this way the infrequent, by which may be meant the non-dominant path code, is separated from the more frequently used

dominant path code, and so does not get loaded into the cache as long as the dominant path is executing.

As Figure 20 of the drawings indicates, a processor chip 200 may operate at a speed
5 of 400 MHz and be associated with an on-board, first level memory cache 202 of
16K. A second level cache 204 of say 512K would be associated with the chip 206.
These are in addition to the normal RAM 208 of perhaps 32 MB operating at a speed
considerably less than the 400 MHz of the first and second level cache memories.
In operation, the processor would pull instructions in from the cache a line at a time
10 (32 bytes). By ensuring that the most frequently used code, ie the compiled dominant
path code, is stored in a separate memory region from the less frequently used code,
the density of the most frequently used instructions in the cache can be increased.
In the process, less frequently used instructions will also be stored together but in
non-cache memory and will thus not pollute the cache.

Dispatch Mechanism for Interface Methods

Technical field

- 5 Optimized execution of object oriented languages which use the 'interface' abstraction, and in particular Java.

Background information

- 10 Java supports single inheritance of class types, with interfaces. Interfaces themselves can be multiply inherited from other interfaces. When a concrete class claims to implement a set of interfaces, it must provide or inherit implementations of every method directly or indirectly defined by those interfaces. See reference [2].
- 15 In object oriented programming, objects are classified in a hierarchical structure with each object associated with attributes (data about its features or properties) and methods (functions it may perform). Typical such functions might be "ring", in the context of a mobile or other telephone, or "play" in the context of audio and/or video reproduction equipment. As one of the features implicit in object-oriented languages,
- 20 such as Java, the features in higher classes of object are "inherited" by objects within the hierarchy of that class.

- For example, as shown in Figure 40, in the class "modes of transport" 400 could be subsumed the subclasses "bike" 402 and "car" 404. The "car" sub-class could be
- 25 subdivided into "saloon" 406 and "sports" 408 and further subdivision is possible according to, for example, the make or model of sports car etc. Certain attributes of the "car" sub-class, such as the number of wheels, model, and so on, will be inherited by the "saloon" and "sports" sub-sub-classes. In similar vein, methods such as "turn on lights" can be common to cars within the hierarchy but in some sub-
- 30 classes the methods themselves may differ to the extent that a certain function has to be performed before lights can actually be turned on. For instance, a sports car with pop-up headlights may need to raise the lights before they can be turned on. In such a case, the inheritance has to be overridden by the need to perform a function before the function in question can be performed.

In another context, the user of a mobile or other telephone may wish to arrange for his handset to emit a different ring depending on whether the call was business or social. In this context "ring" would be termed an "interface". Its significance is that "ring" is a function that a variety of objects in the hierarchy would perform (like
5 "turn on lights" in the car example above) but the actual implementation would differ from object to object. Interfaces therefore cut across hierarchies. An interface is thus a list of functions that the object can perform (such as "ring" or "play" or "record" and so on).

- 10 Single inheritance is usually implemented using dispatch tables (otherwise known as virtual function tables). A subclass inherits the dispatch table of its superclass, extending it with any new methods, and replacing entries which have been overridden.

Multiple inheritance in languages such as C++ is normally implemented using
15 multiple dispatch tables and offsets (see reference [1] for details).

The relevant data is stored in slots in a dispatch table illustrated schematically in Figure 41. The attributes of an object in a table 410 are always located at the same distance from the start of the table. The table includes a pointer 412 to a similar
20 table of methods 414 which are always at the same distance from the start for the same function. However, when interface methods are used (or perhaps when inheritance is overridden), as explained above, there is no longer any certainty of knowing in which slot of the dispatch table the particular function appears. This is a problem peculiar to the Java object oriented language.

25

Problems with known techniques

Up to now, the whole of the dispatch table had to be interrogated to check that the method accessed was the proper method. It had been realised that, ideally, a unique
30 identifier would be needed for the interfaces, but in practice the table cannot be of such a size that everything within it has a unique identifier.

Reverting to the "play" function analogy, there would be one dispatch table for eg video recorder and one for tape recorder. Each would have different interface

references, so "play" might be at position 2 for video recorder and position 22 for tape recorder, say.

5 The logical definition of invoking an interface method is to search the list of methods implemented directly or indirectly by the given class of object. This is clearly slow. This can be improved by searching a 'flat' structure which mirrors the dispatch table.

10 Reference [3] describes an optimization where the last offset at which the interface method was found is remembered, and tried as a first guess next time the invoke interface is encountered. If the guess turns out to be wrong, a fuller search is performed. This approach is based on the assumption that a given call site will tend to operate on the same type of objects.

15 Even if the guess is right, the destination method has to be checked to confirm that it is. In the cases where the guess is wrong, a fairly slow search is needed.

Another approach would be to use an analog of the way C++ multiple inheritance is supported.

Solution to the problems

The solution to this problem, according to the invention in its broadest aspect, is to use an extra level of indirection through a hash table.

5

For the majority of cases where there is no clash in the hash table, invoking an interface is only slightly slower than a standard virtual dispatch, and faster than the known techniques for invoking interface methods. It is also expected to be more compact than the C++ multiple inheritance approach, especially when dispatch table slots contain more than one word of information.

10

Where there is a clash in the interface hash table, a fallback slot in the dispatch table performs the slow but sure search.

- 15 According to other aspects of the invention, the problem of fast access to the required information is solved or alleviated by the use of an interface hash table as well as a dispatch table for each of the various devices.

Detailed description of the invention

20

The inventions of this patent application are:

1. Using a hash for interface methods

Each interface method is allocated a small hash value. This interface hash value can be derived in many ways, but must not exceed the size of the hash table used below.

- 25 It is best if the hash values are chosen to reduce conflicts between interface methods, therefore hash values should be chosen so that methods of the same interface or related interfaces have unique hash values. Clearly an object which implements many interfaces or interfaces with many methods may not be able to avoid clashes.

Naturally, a larger hash table usually reduces the number of clashes.

2. Indirect through a hash table when invoking interface methods

When each concrete class is defined, the set of methods it implements is known, and a dispatch table is created. The dispatch table takes into account methods implementations inherited from its superclass.

- 5 A fixed size hash table is created for each class which maps the interface method hash value described above to a dispatch table index of the corresponding implementation. Where a class implements two or more interface methods which have the same interface hash value, the hash table is set to contain the dispatch table index of the fallback routine described below.
- 10 This hash table is either included at the beginning of the dispatch table, or referenced from the dispatch table.

To invoke an interface method on a given object (in a register),

- a. Load the address of the interface hash table for the given object.
- b. Get the slot number for the specified interface method using its hash as an index
- 15 into the interface hash table.
- c. Load a unique identifier for the destination interface method into a register.
- d. Given the dispatch table slot number, perform a normal virtual invoke.

3. Fallback dispatch table entry

- Where there is a clash between interface method hash entries for a particular class,
- 20 the interface hash table contains the dispatch table index of a fallback method. The fallback method has access (in registers) to the destination object, and a unique identifier for the interface method.

It performs the standard search for that object's implementation of the interface method.

- 25 It will be known to those of skill in the computing art that a hash table is a means of reducing to manageable proportions a data set where information is sparsely populated and there is otherwise a high degree of redundancy within the data set. A hash table thus can reduce the scale of a whole application and thereby reduce the footprint of the device, one of the important features of Java. Overflows are taken into account
- 30 in a way which is already known in the utilisation of hash tables.

Also according to the invention, therefore, a computer system comprises one or more dispatch tables for storing data containing methods appropriate to objects in a class hierarchy and an interface hash table pointing to the location in the dispatch table where a method of interest is located.

- 5 The invention also provides a method of operating a computer system which uses dispatch tables containing methods appropriate to objects in a class hierarchy, comprising the steps of: directing a call for a method to the dispatch table; passing on the call to a hash table containing information as to the location of methods in the dispatch table; and redirecting the call from the hash table to that location in the
10 dispatch table where the method is stored.

- The invention also provides a computer system comprising means for storing data relating to an object, means for calling data relating to a method appropriate to said object, a dispatch table adapted to contain data relating to at least one said method, means for passing the call on to a hash table containing information as to the location
15 of method(s) in said dispatch table and means for redirecting the call from the hash table to the dispatch table to access the location of the called method.

In one form of the invention, there is one interface hash per dispatch table. In another form of the invention, there is a single interface hash table for all the dispatch tables.

- 20 Alternatively, the invention provides both a method of improving the performance of interface dispatching by using a hash table and a computer system comprising a hash table to improve the performance of interface dispatching.

In another aspect, the invention provides a method or a computer system in which the interface reference for a particular method is found by means of a hash table.

- 25 It will be understood that "interface dispatching" is the method by which the slot location for a particular method, eg the slot location number (2) for the "play" function of a video recorder, is located and then the relevant data is called.

Chief advantages of the invention may include faster interface dispatching and/or a reduction in the size of footprint.

In each case, the method or computer system of the invention as specified in the preceding paragraphs may be applied specifically to Java.

The operation of the system can be looked at in another way. Thus, in Figure 42 of the drawings, the data for an object within a particular hierarchy is located in a data structure such as a table 420. The data structure will contain a header and a plurality of frames containing relevant data. When a call is made for a relevant method stored in slots in a dispatch table 422, because of the uncertainty in knowing the exact slot in which that method is located, the dispatch table 422 will automatically re-route the call to a hash table 424 containing a condensed version of the method locations in the dispatch table. Also, because the locations within the hash table are always the same for each method, the hash table will be able to generate an index pointer 426 leading to the correct location in the dispatch table more quickly than searching all possible locations within the dispatch table.

In the event of a clash in the hash table, perhaps because the same location is needed for two interface methods, or perhaps due to being called by two different threads in a multithreaded environment, the hash table will point to a method designed to "sort out" the class and direct the caller to the appropriate location or locations.

Other information

Related Patents:

20 US 5367685

References:

- [1] "The Annotated C++ Reference Manual" by M.Ellis and B.Stroustrup, Addison Wesley (ISBN 0-201-51459-1) pages 217-237
- [2] "The Java Programming Language" by K.Arnold and J.Gosling, Addison
25 Wesley (ISBN 0-201-63455-4) chapter 4
- [3] "The Java Virtual Machine Specification" by T.Lindholm and F.Yellin, Addison Wesley (ISBN 0-201-63452-X) pages 258-260, 403-405

Java is a trademark of Sun Microsystems

Return Barriers - Minimising blocking while a thread's stack is being inspected by a concurrent garbage collector

Introduction

5

In certain computer systems, such as shown schematically in Figure 50, data is stored in (activation) frames 500 in a stack 502 with the most recent activity being regarded as stored in the lowermost frame in the stack (although it could equally be in the uppermost). Data not stored in activation frames are stored in cells in the heap 508.

10 Garbage collection involves tracing the connectivity of all cells. Any that are not traced in this way are therefore invisible and cannot contain any information of relevance. Those cells can thus be released for re-use in the system.

15 In the tracing process, all of the pointers or references in each frame of the stack need to be looked at. For that to happen, it has been necessary up to now for a thread to be paused while tracing is carried out through the whole of that thread's stack. That in turn requires the garbage collection process to be halted while it waits for the thread to give permission for it to interrogate its frames.

20 In the present invention, it has been recognised that the thread need only be paused for as long as it takes to examine the most recent (i.e. the most active) activation frame 504 on that thread's stack. The frame is checked for references or pointers and the return address is edited by substituting for the previous return address the return address of a special code (the "return barrier" code). The special code links the old
25 and the new return addresses to that frame. If the garbage collector is operating on that frame at the time that it is to be returned to, the return barrier code prevents corruption of the data in that frame by pausing return until such time as the garbage collector has moved on to another frame.

30 Technical field

- General: Run-time Environments.
- Specific: Automatic dynamic memory management.

Background information

Any references contained in a thread of control's activation stack need to be treated as part of a tracing Concurrent GC's root set, and need to be examined during the GC process. It is vitally important that the thread being inspected does not alter any
5 information that the GC thread could be examining. One way of achieving this is to suspend execution of the thread to be inspected, allow the GC to inspect the entire contents of the stack, and then to resume execution of the inspected thread.

Problems with known techniques

10 The main problem is that the amount of time a thread will be suspended is determined by the size of the thread's stack, and suspending a thread for too long will lead to noticeable pauses. The technique described by this patent allows a thread to continue execution provided that it is not trying to use a portion of the stack that the GC thread is interested in.

15

Solution to the problems

Suspend the non-GC thread's execution only for as long as it takes to examine the top-most activation frame, and to edit the frame's return address to refer to some special code. Then allow the thread to continue execution while successive caller's
20 activation frames are examined. Once examination of a particular frame is completed, before moving onto the next, edit the frame's return address to refer to the same special code mentioned earlier.

The special code intercepts attempts to return from an activation frame back to the
25 caller's frame. If the caller's frame is currently being examined by the GC thread, compel the non-GC thread to wait until the GC thread has moved onto another frame.

According to a first aspect of the invention, there is provided a method of improving the concurrent garbage collection of reference data contained within a thread stack in
30 a computer system, wherein the thread is only paused for the purpose of garbage collection for the time it takes to examine the current activation frame, rather than the entire stack.

A method according to the previous paragraph may include the step of preventing the

return of an outstanding procedure call into an activation frame whose contents are currently being inspected by the garbage collector until such time as the garbage collector has completed the inspection of that frame.

- 5 Analogous apparatus may also be provided within the scope of the invention, comprising a garbage collector and means for pausing the thread for the purpose of garbage collection only for the time it takes to examine the current activation frame, rather than the entire stack.
- 10 In another aspect, the invention provides a method of improving concurrent garbage collection in a thread stack of a computer system, comprising the steps of: enabling the garbage collection thread to access the thread of interest in the stack; suspending the execution of the thread of interest only for as long as necessary for the most active activation frame to be examined; editing the return address of said frame to the
- 15 return barrier code; allowing the thread of interest to continue execution while successive activation frames are examined; and editing the return address of each said frame to the same return barrier code before moving on to the next frame.

- In a preferred version of the invention, the barrier code is used to prevent the return
- 20 of an outstanding procedure call into an activation frame whose contents are currently being inspected by the garbage collector until such time as the garbage collector has completed the inspection of that frame.

- The invention thereby achieves the objective of reducing the time that the thread of
- 25 interest is suspended. It also maximises the degree of concurrency in a garbage collection system and improves the illusion of concurrency.

Detailed description of the invention

- Let *gcf* be a system-wide global variable which contains a reference to the activation
- 30 frame currently being inspected by the GC thread. Only the GC thread can alter this, but it can be read by any thread.

The GC thread examines a thread *t* thusly:

```
suspend t
let gcf be t's top-most frame.
inspect the contents of frame gcf.
alter gcf's return address to point to barrier
5      intercept code.
let gcf be gcf's caller's frame.
allow t to resume execution.
while gcf is not NULL do
      inspect the contents of frame gcf.
10     alter gcf's return address to point to barrier
           intercept code.
           let gcf be gcf's caller's frame.
endwhile
```

The barrier interception code traps attempt to return to a caller's frame *cf*:

```
15
while cf == gcf do
      wait a short time
endwhile
```

20 Once the GC thread's point of interest has moved on, the non-GC thread can allow its return to caller to complete safely.

It is possible that return barriers established by earlier thread inspections could still be intact on subsequent inspections. In that case the GC thread should not try to establish a barrier if one is already present.

Commercial prospects for the invention

25 This technique will allow a Concurrent GC implementation to behave more "smoothly" than one which forced each thread to be suspended while its entire stack were examined. This will enhance the predictability of the GC implementation, which is a major topic of concern.

Stack walking to allow efficient compiled code deletion in the multi-threaded environment of a unified stack virtual machine

- 5 Where a computer system has finished using memory which it has taken to perform a particular function it is clearly in the interests of speed and efficiency that the used memory is returned as soon as possible for further use. The invention finds particular application in the environment of a unified stack virtual machine in which stack walking allows compiled code to be deleted.

10 **Technical field**

This invention applies preferably to virtual machines where compiled portions of the code being run in the virtual machine appear and need to be removed at various times in the execution of the virtual machine; for example in a dynamically compiling virtual machine.

15 **Background information**

- When executing code using a virtual machine, it is advantageous to produce a compiled version of some or all of the emulated code, and is sometimes desirable or necessary to subsequently remove some or all of these compiled versions. Also, it is advantageous to use a single stack to support the stack requirements of both the emulated machine and also the needs of the virtual machine code itself, and to use a native call instruction to perform the equivalent of an emulated call (invoke) and use of a native return instruction to perform the equivalent of an emulated return in the code being run on the virtual machine.
- 20

Problems with known techniques

- 25 Currently known techniques for virtual machines would require that one or more of the optimizing techniques listed in the background information section be not taken advantage of, or require explicit checks to be used which impair the efficiency of the

system. The problems arise because the use of a native call instruction (or equivalent) to emulate a call or invoke in the virtual machine; this typically leaves the address where execution is to continue once the called method is complete (the "return address") on that stack for that thread, at or near the stack point when the call or
5 invoke is performed. If the native call instruction is part of a compiled version of a section of code, then the return address will point into the compiled version. This causes no problems until the point of deletion of the compiled version; the return address cannot be left pointing to where the compiled version used to be, or some check must be performed at each place where a return is about to be performed to
10 ensure that it is safe to perform a return operation.

Solution to the problems

The solution to these problems is to, at the point of deletion of the compiled code, perform an examination of the virtual machine, looking for cases where a return address exists in these stacks that points to a position within the piece of compiled
15 code to be deleted, and to re-arrange the thread's stack contents to allow seamless continuation of execution of that thread without the compiled version of the code which is about to be deleted. The mechanism is arranged such that the cost of the operation is borne at the time of deletion, with little or no extra cost at normal call/return time, since the relative frequency of the two situations is such that there
20 are many more call/return operations than code deletion operations.

Accordingly, the invention in one aspect provides a method of deleting compiled code in a computer system, comprising:

examining each frame of each stack of each thread in the system;

25 identifying whether a return address points to a portion of compiled code which is to be deleted; and

rearranging the contents of each stack containing the return address so as to enable that thread to continue execution without that portion of the compiled code about to be deleted.

Preferably, the computer system is configured as a Virtual Machine.

In another aspect, the invention provides a computer system comprising means for deleting compiled code, further comprising means for examining each frame of each stack of each thread in the system, means for identifying whether a return address points to a portion of compiled code which is to be deleted, and means for
5 rearranging the contents of each stack containing the return address so as to enable that thread to continue execution without that portion of compiled code about to be deleted.

Detailed description of the invention

10 At code deletion time, each thread in the virtual machine is paused in turn, and the stacks of these threads are scanned, looking for return address values which point at code which is to be deleted. Once one of these cases is found, the state of the stack around the return address value is adjusted to "clean up" the virtual machine state for that thread at the point where the return is encountered (i.e. some time in the future
15 for that thread), and the return address value itself is adjusted to cause the flow of execution to transition to one of a small number of central pieces of code. These centralized pieces of code (termed "glue code") perform some generalized checks and cause the continuation of the flow of execution for that thread in the appropriate manner; usually this will involve interpretation of subsequent emulated instructions
20 until a section of emulated instructions is encountered for which there is a compiled version.

Grey Packets: Low-contention Grey Object sets for Concurrent Marking Garbage Collection in a highly multi-threaded environment

5

Technical field

- **General:** Run-time Environments.
- **Specific:** Automatic dynamic memory management

10 **Background information**

Garbage Collection (GC) is a process whereby a run-time environment can identify memory which was in use at one time, but is now no longer in use, and make it available for re-use for other purposes. Concurrent GC is a way of implementing GC such that other activity in a program or system does not need to be impeded by ongoing GC activity.

15

Tracing GCs (concurrent or otherwise) work by following references, indicated as arrows 700 in Figure 70, between memory items generally indicated as 702, starting from some given root set 704, to establish the set of all items which must be treated as "live". Items which are not in that set are deemed to be "dead" and their memory space can be recycled.

20

The state of the tracing process at any given time can be summarised using the Tricolour Abstraction. Each item has a colour associated with it:

- **White:** This item has not been encountered yet during the tracing process.

25

- **Black:** The item and all the items it refers to have been encountered by the tracing process.

- **Grey:** The item itself has been encountered, but some of the items it refers to may not have been visited.

Any tracing GC algorithm works as follows:

```
5      initially, colour all items white
      recolour grey all items immediately reference from the root
      while grey items exist do
          let g be any grey item
          recolour g black
10     for each item o referenced by g, do
          if o is white then
              recolour o grey
          endif
          endfor
15     endwhile
```

Once this algorithm is complete, the space occupied by any **white** items can be re-used.

Marking GCs tend to implement this abstraction fairly literally, while *copying* GCs do not, with an item's colour implicitly determined by its absolute location in memory. This invention only pertains to marking GC algorithms and techniques.

Efficiency considerations dictate that the set of **grey** items can be treated as a discrete entity that can be added to (by recolouring **grey**) or be removed from (by recolouring **black**). This set is typically implemented as a *stack*. Usually the grey stack tends to
25 be an explicit stack.

In a concurrent GC algorithm, other parts of the system can be altering items while the GC is still tracing. Unless care is taken live objects can be misidentified as dead. A typical way of eliminating this problem is to use a *write barrier* on all operations that could alter the contents of items. Different implementations can work in different
30 ways, but they all tend to require that non-GC threads of control can alter the set of **grey** objects.

Problems with known techniques

The set of **grey** items is a resource shared amongst several threads of control, all of which could alter it. Hence any alteration must be policed by a locking mechanism of some kind. The **grey** set is used heavily during the tracing process,
5 so there is a high probability that any attempt to gain access to the **grey** set will find it already in use. In addition, any overheads incurred by the locking mechanism will tend to be magnified.

Solution to the problems

Instead of having a single monolithic grey item set which has to be locked as a whole on each access, divide the set into discrete segments, or packets, (see for example 706 in Figure 70), preferably such that each thread can be apportioned a
5 segment it (and only it) can work on in isolation. This will minimise the amount of locking required to the occasions when a thread finishes with one packet and needs another to work on.

Some GCs move objects in memory. The system used here does not because of the difficulty of doing so in a concurrent GC. Instead, a "mark and sweep" operation
10 is performed. Here, everything white is released at the end of the tracing or "mark" process. Subsequent to the tracing process there is the sweep phase. In the sweep phase what is black is made white and what is white is made available for future use.

The invention can be regarded as relating to the management of the grey queue or stack in order to overcome the problem that there is a lot of contention for access to
15 the grey stack.

Accordingly, the invention provides, in one aspect, a method of operating a concurrent garbage collecting system in a computer system environment, wherein the grey queue is divided into packets, each packet being accessible by at most one thread at any given time.

20 The principal advantage of the invention is thereby to improve the performance of a GC in a very heavily used system

An additional advantage is to release memory no longer in use.

In another aspect, the invention provides a method of operating a concurrent garbage collecting system in a computer system in a multi-threaded environment, so as to
25 release memory no longer in use, comprising:

tracing the state of each item in a memory group;

allocating an identifier according to whether the item has not yet been encountered during the tracing process (white), the item and all items to which

it refers has been encountered by the tracing process (black), and the item itself has been encountered but some of the items it refers to have not yet been visited (grey);

5 dividing the set or sets allocated with the grey identifier into discrete packets;
and

assigning a respective said packet to each of the threads such that each said thread can work on its respective packet independently of the other said thread(s) and packet(s).

10 The invention also provides a computer system comprising means for operating a concurrent garbage collection system and means for dividing the grey queue into packets such that each packet is accessible by at most one thread at any given time.

In a preferred form of the method or the system, the packets are dynamically managed in that they can be created or destroyed as required.

Detailed description of the invention

15 A set of *grey packet* blocks 706 exists within the program or system. Each block contains a fixed number of slots 708 (each capable of describing a single item reference), and an indication of how many slots are currently in use within that block. Each grey packet is either *checked out*, in which case it is currently being used by one (and only one) particular thread of control, or *checked in*, in which
20 case no particular thread of control is using it. These grey packets are managed by a separate module within the program or system, the *Grey Packet Manager*, or GPM. The GPM offers the following fundamental services:

- `getEmptyPacket()`: obtain an empty packet (or partially filled packet, but *not* a full packet) from the set of checked in packets,
25 alter its status to checked out, and return it to the calling thread.
- `getFullPacket()`: obtain a full packet (or partially filled packet, but *not* an empty packet) from the set of checked in packets, alter its status to checked out, and return it to the calling thread. Return

NULL if only empty packets are present.

- `submitPacket(p)`: Verify that grey packet p is currently checked out, and then alter its status to checked in.

The GPM performs each of the above operations under lock.

- 5 Each thread of control (including the GC) has a *packet-in-hand* (or *tl-pih*) grey packet pointer. This pointer may be NULL (indicating that the thread has no packet in hand), but if non-NULL it must refer to a checked out packet.

M a r k i n g a n i t e m i a s g r e y b e c o m e s :

```
10     if tl-pih is NULL then
             tl-pih = getEmptyPacket()
         else if tl-pih is full then
             submitPacket(tl-pih)
             tl-pih = getEmptyPacket()
15     endif
         recolour  $i$  grey
         insert  $i$  into tl-pih
```

The main blackening algorithm becomes:

```
20     obtain a packet  $p$  to blacken
         while  $p$  is not NULL do
             for each reference  $g$  in  $p$ 
                 recolour  $g$  black
                 for each item  $i$  referenced from  $g$  do
25             if  $i$  is white then
                     mark  $i$  as grey
                 endif
             endfor
             remove  $g$  from  $p$ 
30     endfor
         submitPacket( $p$ ) // now empty
         obtain a packet  $p$  to blacken
```

endwhile

Obtaining a packet to blacken is:

```
5      if tl-pih is not NULL then
          let p be tl-pih
          tl-pih = NULL
      else
          let p = getFullPacket()
      endif
```

- 10 The idea is that both the marking and blackening processes most operate only on the thread's packet in hand; which if present at all can be guaranteed not to be visible to any other thread. Hence, most of the time no locking is required, except when interaction with the GPM is required to submit packets, obtain empty packets or packets to blacken.
- 15 Periodically each non-GC thread must submit any packet in hand back to the GPM (only the GC can blacken packets). This is typically done when the GC needs to examine a non-GC thread's local datastructures.

Commercial prospects for the invention

- 20 The primary aim of this technique is improve the performance of Concurrent GC in highly multi-threaded environments, by virtue of minimising locked accesses to a global datastructure. Hence a commercial product utilising Concurrent GC with this technique will perform better than one using a more traditional approach.

Other information

- 25 A general summary of GC technology, Concurrent and otherwise, can be found in "Garbage Collection: Algorithms for Automatic Dynamic Memory Management" by Richard Jones and Rafael Lins, published by John Wiley, ISBN 0-471-94148-4. The disclosure of this document is hereby incorporated by reference.

Executing Device Driver Interrupt Handlers Written in Java

Technical field

This invention belongs to the virtual machine technical field.

5 Background information

In the implementation of device driver software, it is usually required that device interrupts be dealt with by code within the driver itself.

The code written as part of the driver to deal with such interrupts usually has significant constraints placed upon it; this is because such code can be executed at
10 almost any time at all in relation to the main-line application, often using a small, fixed-size, separate stack provided by the operating system.

This invention relates to the full implementation of a device driver including its interrupt handlers in Java (although the many parts of the invention would certainly apply to other interpreted languages as well).

15 The fact that Java is a garbage-collected language adds to the complexity of this problem in that interrupt handler code may need to run successfully at any arbitrary point in the garbage collection process without interfering with it or failing itself in some way due to it.

Problems with known techniques

20 The only prior technique the applicant is aware of for handling this problem involved the (Java) Virtual Machine (VM) having its own dedicated interrupt handlers implemented in a non-interpreted language (Assembler or C) which handled the interrupt in a generic way and then dismissed it before passing a note of its occurrence to a high priority java thread running at non-interrupt-level.

25 There are two main problems with such a technique, both stemming from the fact that the Java code written to handle device interrupts is significantly divorced from real

interrupt level; firstly, the problem that special device handling (reading/writing special values from/to device registers) may no longer necessarily be valid, according to the type of device in question. The real interrupt has already been dismissed; secondly, a substantial length of time may have elapsed between the real interrupt occurring and
5 the Java code to handle it actually executing. The problem with the prior art is that it cannot handle interrupts in real time. For example the SUN prior art system just makes a request to process the interrupt in a normal thread as soon as possible.

The fact that Java code never runs at real interrupt level in this solution does, however, substantially alleviate problems with garbage collection.

10 **Solution to the problems**

The solution to the problem of actually getting Java code to run at real interrupt level was broken down into sub-problems, solved as follows; in the following any, some or all of the sub-problems and any of the solutions may be combined in any appropriate way:

15 **Sub-problem:** Real interrupt level runs on a small, separate OS-supplied stack which is unsuitable for use by the Java bytecode execution engine.

Solution: Have a normal Java thread stack ready and waiting to be switched to when an interrupt occurs.

Sub-problem: the bytecode execution engine must not attempt any potentially
20 blocking synchronization operations while executing the bytecode of an interrupt handler.

Solution: ensure that the normal routes through the bytecode execution engine have no potentially blocking synchronization operations - this is desirable from a performance point of view anyway! Additionally, make sure that the nature of the
25 bytecode of an interrupt handler never requires other than the normal routes through the bytecode execution engine.

Sub-problem: the bytecode execution engine must not do anything that could interfere with or fail because of any phase of garbage collection occurring (potentially

simultaneously) at non-interrupt level.

Solution: this is achieved basically by denying interrupt level code the full flexibility of the garbage collected Java heap.

Sub-problem: the solutions of the last two sub-problems would seem to indicate that
5 communication between interrupt-level Java and non-interrupt-level Java is hard, if not impossible!

Solution: a special mechanism is made available to the Java application programmer to enable the passing of information from the Java code which runs at interrupt level to the rest of the application.

10 In one of its most general aspects, the invention comprises a computer system or a method of operating a computer system in which a normal Java thread stack is provided and kept ready and waiting to be switched on when an interrupt is detected.

In its preferred form, the invention lies in the context of a software VM and the significant feature of the invention is that the system or method runs Java bytecode at
15 interrupt level.

In another aspect, the invention provides a method of implementing device driver interrupts in a computer system structured as a virtual machine, the method comprising having a special interrupt stack ready to run the instant an interrupt call is received.

In a preferred form of the invention as set out in the preceding paragraph, the system
20 is ready to run an interpreted language (eg the Java code). In a modification, the special interrupt thread stack is a normal (Java) thread stack which is switched in when an interrupt occurs;

The invention also provides a method of implementing device driver interrupts in a computer system structured as or implementing a virtual machine, the method
25 comprising preventing the bytecode execution engine from attempting any potentially blocking synchronisation operations while executing the bytecode of said interrupt handlers.

The invention further provides a method of implementing device driver interrupts in

a computer system structured as or implementing a virtual machine, the method comprising preventing the bytecode execution engine from interfering with simultaneous garbage collection at non-interrupt level.

5 The invention yet further provides a method of implementing device driver interrupts in a computer system structured as or implementing a virtual machine, the method comprising enabling information from the (Java) code running at interrupt level to pass to the rest of the application.

The invention has the advantage of enabling interrupt handler code to run successfully at any point in the garbage collection process without interference.

10 The invention also extends to a computer system provided with means for implementing device driver interrupts, comprising a special interrupt stack ready to run the instant an interrupt call is received.

15 In a preferred form of the invention as set out in the preceding paragraph, the system is ready to run an interpreted language (eg the Java code). In a modification, the special interrupt thread stack is a normal (Java) thread stack which is switched in when an interrupt occurs.

20 The invention further extends to a computer system provided with means for implementing device driver interrupts, comprising means for preventing the bytecode execution engine from attempting any potentially blocking synchronisation operations while executing the bytecode of said interrupt handlers.

The invention yet further extends to a computer system provided with means for implementing device driver interrupts, comprising means for preventing the bytecode execution engine from interfering with simultaneous garbage collection at non-interrupt level.

25 The invention yet further extends to a computer system provided with means for implementing device driver interrupts, comprising means for enabling information from the (Java) code running at interrupt level to pass to the rest of the application.

The invention has the advantage of enabling interrupt handler code to run successfully

at any point in the garbage collection process without interference.

Detailed description of the invention

The details of how the sub-problems mentioned earlier were solved are as follows:

Sub-problem: Real interrupt level runs on a small, separate OS-supplied stack which
5 is unsuitable for use by the Java bytecode execution engine.

Solution: Have a normal Java thread stack ready and waiting to be switched to when an interrupt occurs; this is achieved by having a normal Java thread created as part of the Java application start-up code partially destroy itself by a call on a special native method, `waitForFirstInterrupt`.

10 The `waitForFirstInterrupt` native method is a fundamental part of the interrupt handling mechanism of the Insignia Java VM, it destroys any O/S related thread components apart from the stack (this stack contains information concerning just *where* in the java application the particular call to `waitForFirstInterrupt` was made from); the location of this stack is registered with the interrupt handling mechanism of the
15 Insignia Java VM for later use with respect to a particular device or interrupt.

When the first interrupt is received from the relevant device, the operating system will enter the interrupt handler of the interrupt handling mechanism of the Insignia Java VM - this will switch stacks to the relevant Java stack preserved earlier and then execute a native method return sequence which will, as always, re-enter the Java
20 execution engine at the location following the native method (as recorded in the stack).

At this point, the bytecode execution engine is executing Java bytecode at O/S interrupt level - this places various constraints upon the bytecode execution engine whilst executing such bytecode such as not attempting any blocking thread synchronization operation, not doing anything that could interfere with or fail because
25 of any phase of garbage collection occurring (potentially simultaneously) at non-interrupt level; these sub-problems are covered later.

At this point it is worth noticing that this solution is completely compatible with the dynamic (or off-line pre-) compilation technology described elsewhere. It is quite

possible that the bytecode being referenced has been compiled into machine code for speedy execution, the native method return mechanism will select the machine code version if present or select the interpreter for bytecode-by-bytecode interpretation.

When the Java code of the interrupt handler has i) interacted with the device using native methods supplied as part of the Insignia hardware access Java package as appropriate to the specifics of the device and interrupt type that has occurred and, ii) interacted with the rest of the Java application involved with the device (the non-interrupt part of the application, that is) through the use of native methods in the Insignia interrupt handling package as appropriate, it must allow the execution of normal, non-interrupt code to continue. This is achieved by calling another special native method, `waitForInterrupt` (as opposed to `waitForFirstInterrupt`, above).

The `waitForInterrupt` native method gets the Java stack ready for a subsequent activation by another interrupt and then switches back to the O/S's small, dedicated interrupt stack and then performs the return appropriate to the particular O/S, allowing it to perform the actions necessary to return to non-interrupt running.

Sub-problem: the bytecode execution engine must not attempt any potentially blocking synchronization operations while executing the bytecode of an interrupt handler.

Solution: ensure that the normal routes through the bytecode execution engine have no potentially blocking synchronization operations - this is desirable from a performance point of view anyway! Additionally, make sure that the nature of the bytecode of an interrupt handler never requires other than the normal routes through the bytecode execution engine.

In the case of Java (as opposed to a more general interpreted language), this latter point means that constant pool entries must be pre-resolved (constant-pool resolution is a process that normally occurs the first time that particular bytecode is executed and can result in many potentially blocking synchronization operations).

In practice, we would probably go one stage further and pre-compile the Java bytecodes of interrupt handlers (although this is not strictly necessary if the dynamic compilation system does not require the interpreter to perform any potentially blocking

synchronization operations as a matter of course).

Sub-problem: the bytecode execution engine must not do anything that could interfere with or fail because of any phase of garbage collection occurring (potentially simultaneously) at non-interrupt level.

- 5 **Solution:** this is achieved basically by denying interrupt level code the full flexibility of the garbage collected Java heap as follows:

10 The special Java thread that comprises an interrupt handler is allowed to allocate objects as part of its start-up phase (before it calls the special native method, `waitForFirstInterrupt`); these objects will persist for the entire life-time of the system (they will never be garbage-collected away). At the time that the Java thread ceases to be normal (just becoming a stack for use at interrupt level as described above), this set of heap objects becomes the set of the only heap objects that the interrupt-level Java code can ever see; in this way, this set of objects is a fixed presence in the java heap that is independent of garbage collection activities; in this way also, the garbage collector running at non-interrupt level can carry on in confidence that interrupt-level Java code can never interfere with its operation (or vice versa) because the interrupt level code will only ever be dealing with its own set of objects.

20 It is permissible for non-interrupt Java code to see references to interrupt Java objects - a crucial thing is that it must not use this as an opportunity to store references to non-interrupt Java objects into these interrupt objects for interrupt Java code to see!

Policing mechanisms can be put into place on development VMs to ensure that this policy is not violated.

25 **Sub-problem:** the solutions of the last two sub-problems would seem to indicate that communication between interrupt-level Java and non-interrupt-level Java is hard, if not impossible!

Solution: a special mechanism is made available to the Java application programmer to enable the passing of information from the Java code which runs at interrupt level to the rest of the application.

Native methods are provided as part of the interrupt package to allow the passing of information from interrupt level to non-interrupt level; this allows the non-interrupt code to be suspended inside a call on the *read* native method and be woken when an interrupt has completed having made a call on the associated *write* method.

Use of Class Loader to allow direct invocation of Non-final Instance Methods

General

- 5 This aspect of the invention is concerned with method inheritance, “method” being a Java term for functions such as “area” of a circle and any of the other functions such as “play”, “turn on lights” etc as already discussed above.

10 In prior systems and methods a call to a method of a given name will cause one of a number of different implementations of the named method to be executed according to which object one is interested in (eg a “play” function in a video recorder, tape recorder etc.) This is called a “polymorphic call”. Under these circumstances, one would compile differently according to the object. Because of these problems one would make no assumptions about the destination method, and so one has to compile the call to it less than optimally.

- 15 Our solution to the problem of optimising the system aims to optimise for the set of currently loaded classes. If another class is loaded which overrides some methods of previously loaded classes optimisation will be changed for calls to methods that the new class overrides, ie if we discover that the method is polymorphic then we go back and undo the specific optimisation assumptions.

- 20 Accordingly, the invention provides, in one aspect, a method of operating a computer system comprising the steps of:

compiling a call to the method for a given class;

determining for a new sub-class whether a method of the class has previously been treated as final; and

- 25 adjusting the compilation of the call to the method for the given class if the method is not final.

The method according to the previous paragraph advantageously takes advantage of the assumption that the method being called is "final".

In a second aspect, the invention provides a computer system comprising means for compiling calls to a method for a given class, means for determining whether the
5 method has previously been treated as final, and means for adjusting the compilation of the call to the method for the given class if the method is not final.

The invention therefore enhances opportunities for optimisation of the computer system.

Technical field

10 This invention applies preferably to virtual machines executing Object Oriented programs, where the classes of the objects in question are dynamically loaded and/or discovered by the virtual machine. It applies particularly to Virtual Machines where some optimizations can be performed if a potentially polymorphic method of a class can be safely assumed to be non-polymorphic; for example in a dynamically
15 compiling virtual machine.

Background information

In an object oriented environment, a method is said to be polymorphic if a number of different implementations of the method are available in the system, and the implementation used is chosen at each point where the method is invoked and at each
20 time that the point of invocation is reached in the execution of the program. This situation typically comes about in object oriented systems due to inheritance, whereby a class (a description of a type of object, including the methods that can be invoked on instances of that class) is taken as a basis for a subsequent class. This new class is termed the subclass of the original class, which is termed the superclass, and is
25 said to inherit all the aspects (including the methods) of the superclass. However, the new subclass can override some or all of the methods inherited from the superclass and provide its own implementations of these methods; these overridden methods are now polymorphic, and the implementation which is chosen to be used in any case where one of these overridden methods is invoked is governed by the class of the
30 object that the invocation is associated with.

One approach to the situation where the system can be affected by classes that are discovered and/or loaded at some time after a class or method or part of a method is converted to a compiled form is to make no assumptions in the compiled version about the method being invoked by the dynamic mechanism. In the Java environment, methods can be marked as "final", which means that it is illegal to override them in subsequent subclasses. This allows assumptions to be made about which method implementation is being invoked, but the majority of methods in typical Java classes are not so marked for reasons of flexibility.

Problems with known techniques

Whilst the approach described above will yield a system which works, a potentially large number of optimization opportunities will be missed, since the cases where a method (if it is not polymorphic at the time that the compilation of a call to the method is attempted) remains non-polymorphic are seen in normal use to predominate. If, however, the assumption is made that the method is not polymorphic, then the system runs into problems if the assumption is later on found to be false, due to a new class being loaded into the system.

Solution to the problems

The problems outlined above are solved by this invention using a number of factors. First of these is the ability to adjust the action of the class loader in this situation to notify the manager of the compiled code when an assumption about the finality of a method previously made during prior compilation is found to be false. The second factor is the ability to, at any time, remove from the system compiled code which is no longer wanted for whatever reason. A third factor is the use of patches to existing compiled code sequences, allowing the code action to be adjusted whilst the code is "live", and being potentially executed by one or more threads of the virtual machine.

Detailed description of the invention

As each section of code is considered by the virtual machine for compilation, any potentially polymorphic invocations out of the code section are also considered. For each such invocation, if the destination of the invoke is fixed at the time of compilation (i.e. there is only one implementation of that method in the system at that

time), then the assumption is made by the compilation system that that situation will continue to be so. This allows various optimizations to be made, including but not limited to the in-lining of the single implementation of the invoked method. Before the resulting compiled version of the code is made available to the rest of the virtual
5 machine as a potential part of the execution environment, the method being invoked is marked in its VM data structure as having compiled code which assumes that it is not overridden.

Subsequently, as each new class is loaded, the class loader checks to see if any of the methods of the new class override a method with the marker set in its data structure.
10 If this is the case, the class loader calls back to the compiled code manager section of the virtual machine and requests that all the effected compiled code is deleted or made inaccessible.

If the compiled version of the calling code is arranged not to make many assumptions about the internal details of the method it is invoking, a simpler mechanism that can
15 be used in parallel with the above mechanism is to allow patching of the compiled version of the calling code to call directly to the compiled version of the method being called. This direct patch can be relatively easily undone if a subsequently loaded class is found to override the method in question using the same detection mechanism as described above. The benefit of the patched version is that it avoids the
20 overheads of making the dynamic decision at the time of the invoke as to which implementation to choose to invoke. Even if there is only one possibility, the overhead is present unless the patched form is used.

Reducing Search times for unordered lists in multi-threaded environment

Technical field

Data structure access in multi-threaded environment.

5 Background information

In a multi-threaded environment, extreme care must be taken whenever shared (i.e. those able to be accessed by more than one thread at the same time) data structures are modified. Without this care, threads may see partially updated data and so maintain a corrupt view of the data structure. A frequent implementation technique is to lock access the data structure with a mutually-exclusive access mechanism, a *Mutex*. Some data structures have many times more accesses that read the data structure than accesses that make modifications, and these benefit from an access mechanism that does not use a mutex for accesses that just *read* the data.

- 15 This invention provides an optimisation in the accessing of unordered, singly linked lists that can be read without mutex. It does not address the problem of inserting new entries into such a list, nor the more difficult problem of removing old entries, but neither does it increase the complexity of either task.

Problems with known techniques

- 20 If the data structure illustrated schematically in Figure 100 is accessed for reading very frequently, and if the access mechanism without any mutex is efficient, then the time taken to acquire and release the mutex may become a significant proportion of the time to access the data structure.

Solution to the problems

- 25 With reference especially to Figures 101 and 102, by making the *list* into a *loop* by the addition of pointer 1002 any thread can independently change the pointer 1004 to the start of the *loop* to indicate the most likely element such as 1006 to be accessed next time. On the subsequent access the item to be searched for has become more

likely to be the first item looked at.

According to the invention there is provided a method of accessing data in a list in a computer system, comprising the steps of: arranging the list in the form of a loop; accessing a given element in the loop; and selecting that element as being the start of the loop for the next access.

The invention also provides a computer system for accessing data in a list, comprising means for arranging the data in the form of a closed loop, means for accessing a given element in the loop, and means for selecting that element as the start of the loop for the next access.

- 10 The principal advantages of the invention are a reduction in access time to the data in the list and the avoidance of the need for a mutually-exclusive access mechanism, otherwise known as a mutex.

The method is particularly advantageous in a multi-threaded environment. The selection is advantageously performed as a single write operation, ie it is atomic.

- 15 This would be of great advantage in a multi-threaded environment if stability were to be maintained.

In a preferred embodiment, the invention can be put into effect without using any *mutex*.

- In a preferred implementation of the technique, each entry in the list refers to a chunk of compiled code.

Detailed description of the invention

- In Figure 100, the terminating entry of the traditional list is designated by a null pointer at node 1006. The list head, 1004, points at the first node, 1010. The invention replaces the null pointer at node 1006 with a pointer to the start of the list, 1010. This creates a cyclic *loop* rather than the more traditional *list*. By implementing the data structure as a loop we have created the property that the list effectively has no natural starting node. Whichever node we choose can be treated as a head-of-list, processing being achieved by visiting all nodes until the start point is again reached.

So whereas we would process a traditional list with:

```
    ptr = list_head;
    while (ptr != NULL) do
5      if (ptr-key == key) then
        return ptr-data
      endif
      ptr = Next(ptr)
    endwhile
```

10 the same effect is achieved when processing a loop by the algorithm:

```
    ptr = list_head;
    first_ptr = ptr;
    if (ptr != NULL) then
15      do
        if (ptr-key == key) then
          return ptr-data
        endif
        ptr = Next(ptr)
20      while (ptr != first_ptr) do
    endif
```

The benefit of the invention is achieved by allowing the *read* access to re-write the *list_head* without mutex. Since any node within the loop can equally validly be treated as the head of the list, provided a thread can atomically update the *list_head*, no
25 mutex is required. That is, if two threads update the *list_head* at almost the same time it does not matter which thread atomically writes first, the data structure always remains consistent.

In any environment where the list is unordered but there is an above average probability that the last item found in a search of the list will also be asked for the
30 next time the list is searched, then by changing the *list_head* as described above we reduce the number of nodes visited in the search, and hence the search time. One example of such an environment is in a virtual machine where a hash-table with overflow chains is used to map between bytecodes in the source Java and any

equivalent compiled host code.

Method for automatic test and verification of dynamically compiled code in a virtual machine

Technical field

5 Virtual machines

Dynamically compiled code for virtual machines

Debugging tools for virtual machines

Background information

10 Errors in dynamically compiled code frequently manifest themselves a long time after the error actually occurred, making it difficult to identify the true cause.

When changing and/or adding optimisations to a dynamic compiler, it is difficult to demonstrate that the code produced as a result is correct.

Problems with other techniques

15 In one embodiment, two execution engines are used within the same process and their results are compared. One execution engine is the trusted implementation (the master) and the other is the implementation under test (the slave). This test process is limited to a singly-threaded application and can be both cumbersome and time-consuming, since the execution engines must be run in series. The process is to save the initial state (state 1), run part of the master, save the final state of the master (state 2),
20 restore state 1, run part of the slave, then check the final state of the slave against the saved state 2 to detect discrepancies.

The applicant is not aware of other techniques except examining the results of specific test code.

25 Techniques for identifying the cause of errors once identified tend to perturb the system under test, often to the extent of changing or removing (temporarily) the failure behaviour. The object of the invention is therefore to provide a quicker and more reliable system and method for testing pieces of executable code produced by

a dynamic compiler.

Solution to the problems

In a first aspect, the invention provides a method of testing one implementation of a particular specification against a different implementation of the same specification,
5 comprising the steps of:

defining corresponding synchronisation points in both implementations; executing the said one implementation and the said similar implementation; and comparing the states produced by both pieces of code at the synchronisation points.

The invention also provides a computer system for testing one implementation of a
10 particular specification against a different implementation of the same specification, comprising means for defining corresponding synchronisation points in both implementations, means for executing both implementations, and means for comparing the states produced by both implementations at the synchronisation points.

Particularly where the specification is of an execution engine for Java bytecode, the
15 two implementations are advantageously built into different virtual machines (VMs). The VM containing the trusted implementation is called the Master VM, and the VM containing the implementation under test is called the Slave VM. Both VMs execute the same application and communicate with each other at known synchronisation points to exchange and compare the state of the virtual machine.

20 The advantages of this method over the embodiment described above include the following possibilities:

- the slave VM undergoes minimal perturbation, reducing the possibility of changing the failure behaviour;
- the state acted on by each implementation is independent of the state acted
25 on by the other;
- the invention is applicable to multi-threaded applications;
- the VMs need not be running on the same machine (nor indeed on the same

architecture or operating system), thus increasing efficiency;

- the implementations run in parallel rather than in series, thus increasing efficiency;

- the Slave VM requires few extra resources for this invention, increasing its applicability.

If a discrepancy is found in the states of the two virtual machines then it will indicate that since the previous synchronisation point the behaviour of the two VMs has differed. The dynamically compiled code which has been executed by the Slave VM since the last synchronisation point can easily be identified.

10 If a discrepancy is found, it indicates that one (or possibly both) VMs contains an error. The error is generally found in the VM under test if only because it is likely to be newer, more complex and less tested than the trusted VM, but nevertheless this method may identify an error in the trusted VM provided that the VM under test is either correct or at least differently incorrect.

15 In preferred forms of the invention, the synchronisation points are selected from:

conditional transfers of control; method/function/procedure calls or returns; backward transfers of control.

Where the master and slave virtual machines support multiple threads, each thread is synchronised independently with the corresponding thread in the other virtual machine.

20 Preferably, the programming language is Java and synchronisation is effected on a per thread basis. More especially, in that case there are preferably a plurality of asynchronously handled thread pairs.

Advantageously in the above system, the synchronisation points may be chosen (at least) in (partial) dependence upon (and preferably in proportion to) the length of code. This gives the dynamic compiler the best chance of performing the same optimisations as when not under test and hence reduces perturbation.

In any of the systems defined in the preceding paragraphs, the system may be part of a virtual machine. The system may operate independent processes and may optionally be concurrent.

Detailed description of the invention

- 5 While this method has been developed primarily for a Java virtual machine, the techniques used are more generally applicable.

Choice of synchronisation points

Both VMs must use the same synchronisation points. A suitable choice could contain all or some of the following:

- 10
- conditional transfers of control
 - method/function/procedure calls
 - method/function/procedure returns
 - backward transfers of control

The choice of synchronisation points is discussed further in the section "The Slave
15 Virtual Machine" below.

If the virtual machine supports dynamically allocated objects, then the Master and Slave VMs must ensure that corresponding objects are identified on each VM.

If the virtual machine supports multiple threads, then the Master and Slave VMs must ensure that corresponding threads are identified on each VM and that each thread is
20 independently synchronised.

If the virtual machine supports native methods or functions (i.e. those which are not executed directly rather than via the virtual machine's execution engine), then most have to be executed solely on the Master and the return values and any necessary side-effects must be transmitted to the Slave. For example, a native function which
25 returns the time of day would always be executed on the Master, while a native function which causes the virtual machine to exit should be executed on both Master

and Slave.

In the case of a Java virtual machine, a native method may effect an invocation on a method written in Java. Regardless of whether the native method itself is being executed on both VMs or solely on the Master, such a Java method must be executed
5 on both VMs.

The Master virtual machine

The Master (trusted) virtual machine is heavily instrumented to record all reads of the virtual machine state and all modifications of the virtual machine state.

Each execution thread synchronises independently with the corresponding execution
10 thread on the Slave VM. The basic synchronisation loop is shown under the heading Per-thread synchronisation loop below.

Per-thread synchronisation loop

	MASTER VM	SLAVE VM
15		
	MasterStart:	SlaveStart:
		(wait for SB message)
20	clear state info database run to next sync point, gathering info on state reads and writes	
25	send SB message to Slave (wait for SA message)	
30		instantiate before values run to next sync point send SA message to Master goto SlaveStart
35	check values against SB message goto MasterStart	

The Master starts its synchronisation loop by clearing its database of state information. It then runs to the next synchronisation point, adding to its state information database when any item of the virtual machine state is read or written. The item's type and value at any read, and before and after any write are saved.

- 5 At the synchronisation point, the Master sends a State Before (SB) message to the slave and waits until it receives the corresponding State After (SA) message from the Slave once the Slave has reached the corresponding synchronisation point. When the Master receives the SA message from the Slave, it checks that all the virtual machine state items written by the Slave since the previous synchronisation point have the
- 10 correct type and value. If any item is incorrect then the error can be communicated to the user immediately or batched for later examination. The Master can then proceed with the next iteration of the synchronisation loop.

- An optimisation to the Master loop would be to have it continue with its next synchronisation loop immediately after sending the SB message rather than waiting
- 15 for the SA message from the Slave. That wait can be postponed until the Master is ready to send its next SB message, in the expectation that the wait would be very much reduced, possibly to zero. A further optimisation would be for the Master to retain a buffer of several SB messages so that it could run several synchronisation loops before having to wait for the Slave. These optimisations may be worthwhile
- 20 since the Master synchronisation loop is likely to be slower than the Slave. The Master execution engine is typically a much slower implementation than the Slave execution engine and in addition is burdened with the majority of the costs of this invention.

The Slave virtual machine

- 25 The Slave virtual machine (the VM under test) must keep its virtual machine state either up to date or easily updateable at synchronisation points, so that the types and values of state items written since the previous synchronisation point can be collected and sent to the Master. It is very important that this requirement is implemented in such a way as to minimise any perturbation to the Slave's usual mode
- 30 of operation. When the Slave contains an optimising dynamic compiler it is particularly important not to generate different code when testing compared to that produced in normal operation.

This can be achieved by a combination of careful choice of synchronisation points to coincide with times when the compiled code is likely to have the necessary state available if not in the correct place, and having the dynamic compiler generate a special piece of code at synchronisation points to save the contents of any state items
5 not yet up to date, update them, create and send the SA message, and finally restore the saved contents of those state items especially updated for the synchronisation point.

Multi-threading issues

If the virtual machine is multi-threaded, then the Master and Slave VMs will
10 synchronise each execution thread separately. They must have a method of identifying corresponding execution threads on both VMs and exchanging messages at critical points such as thread and monitor state changes and creation.

The communication mechanism

The communication required for this method can be implemented on top of any
15 suitable transport mechanism e.g. sockets or named pipes.

General Considerations

A specific example of a preferred embodiment of virtual machine is now described with reference to Figure 200.

- 5 The virtual machine 2000 is an executable code installed in the particular item of equipment 2002. It can provide a degree of independence from the hardware and operating system. The virtual machine may typically comprise any, some or all of the following features: an operating engine, a library of routines, one or more interpreters, one or more compilers, storage means for storing a plurality of
10 instruction sequences, queue management means, and buffer management means.

The virtual machine is coupled to one or more applications 2004 on one side (the "high level" side), and, on the other side (the "low level" side), perhaps via various intermediate logical units, to the hardware 2006 of the item of equipment. The hardware can be regarded as including various ports or interfaces 2008 (perhaps an
15 interface for accepting user input); the virtual machine receives events from those ports or interfaces. The hardware also includes one or more processors/control means 2010 and memory 2012.

The following considerations apply to any and all the inventions and aspects of the inventions described above.

- 20 In any or all of the aforementioned, certain features of the present invention have been implemented using computer software. However, it will of course be clear to the skilled man that any of these features may be implemented using hardware or a combination of hardware and software. Furthermore, it will be readily understood that the functions performed by the hardware, the computer software, and such like
25 are performed on or using electrical and like signals.

Features which relate to the storage of information may be implemented by suitable memory locations or stores. Features which relate to the processing of information may be implemented by a suitable processor or control means, either in software or in hardware or in a combination of the two.

- 30 In any or all of the aforementioned, the invention may be embodied in any, some or all of the following forms: it may be embodied in a method of operating a computer

system; it may be embodied in the computer system itself; it may be embodied in a computer system when programmed with or adapted or arranged to execute the method of operating that system; and/or it may be embodied in a computer-readable storage medium having a program recorded thereon which is adapted to operate
5 according to the method of operating the system.

As used herein throughout the term "computer system" may be interchanged for "computer", "system", "equipment", "apparatus", "machine" and like terms. The computer system may be or may comprise a virtual machine.

In any or all of the aforementioned, different features and aspects described above,
10 including method and apparatus features and aspects, may be combined in any appropriate fashion.

It will be understood that the present invention(s) has been described above purely by way of example, and modifications of detail can be made within the scope of the invention.

15 Each feature disclosed in the description, and (where appropriate) the claims and drawings may be provided independently or in any appropriate combination.

THIS PAGE BLANK (USPTO)

Fig 10

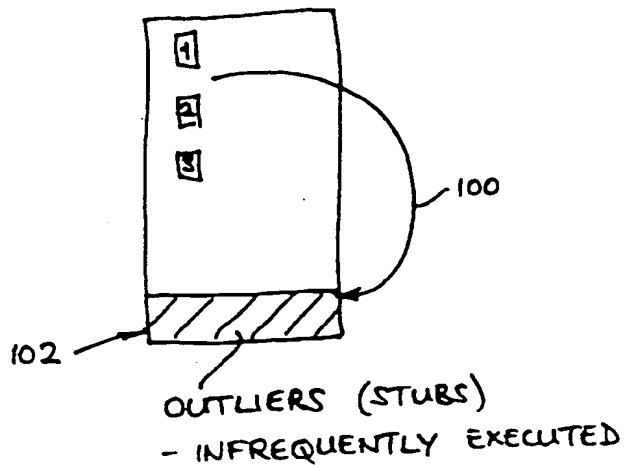
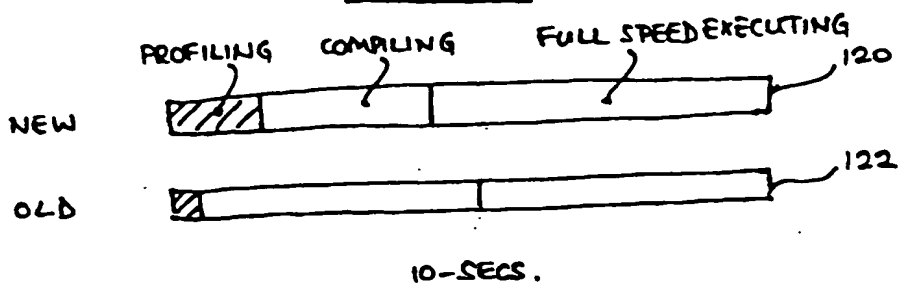


FIG 12



THIS PAGE BLANK (USPTO)

FIG 20

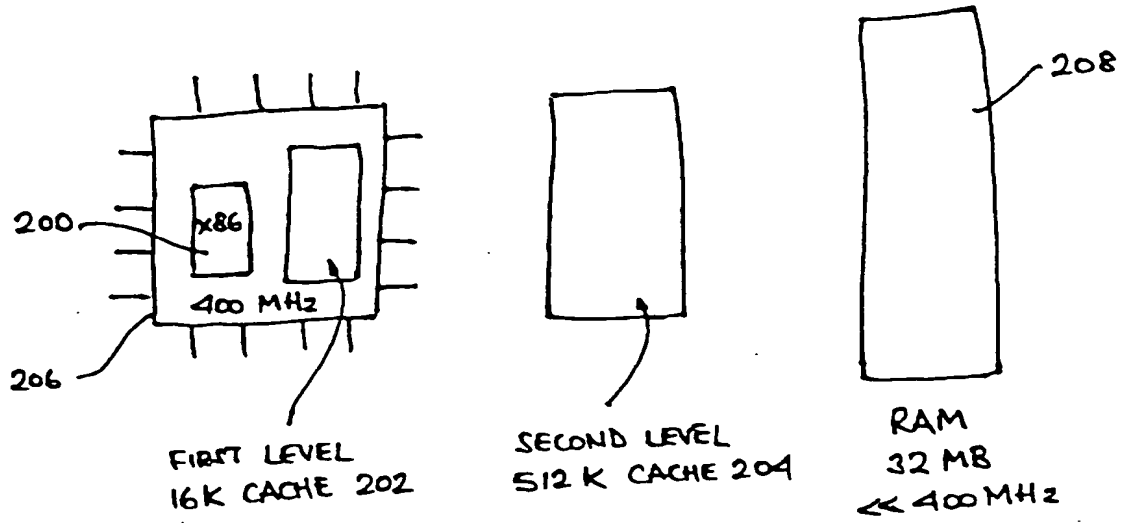
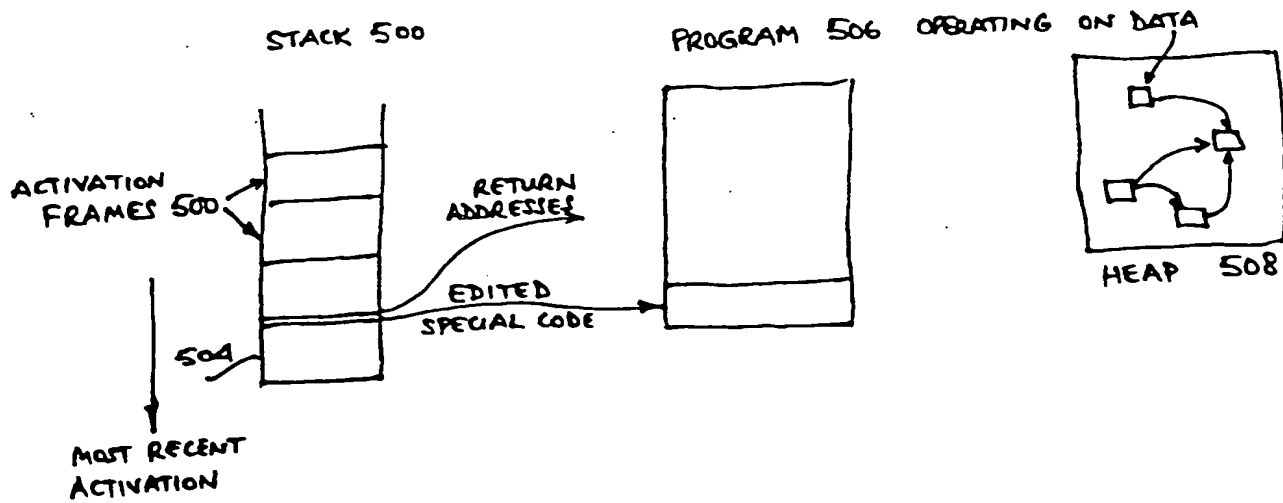


FIG 50



THIS PAGE BLANK (USPTO)

FIG 40

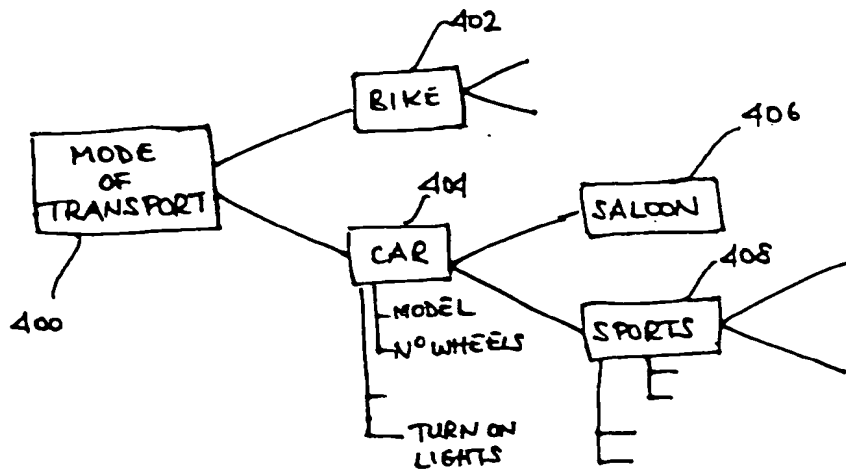


FIG 41

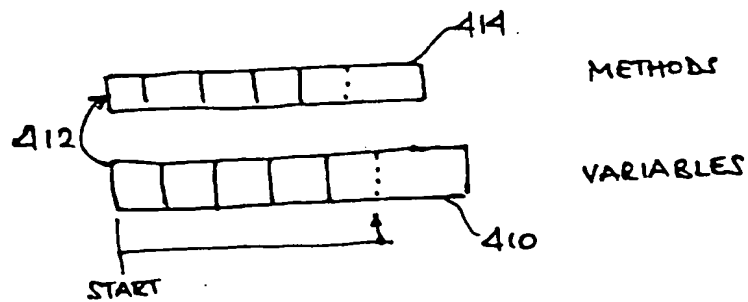
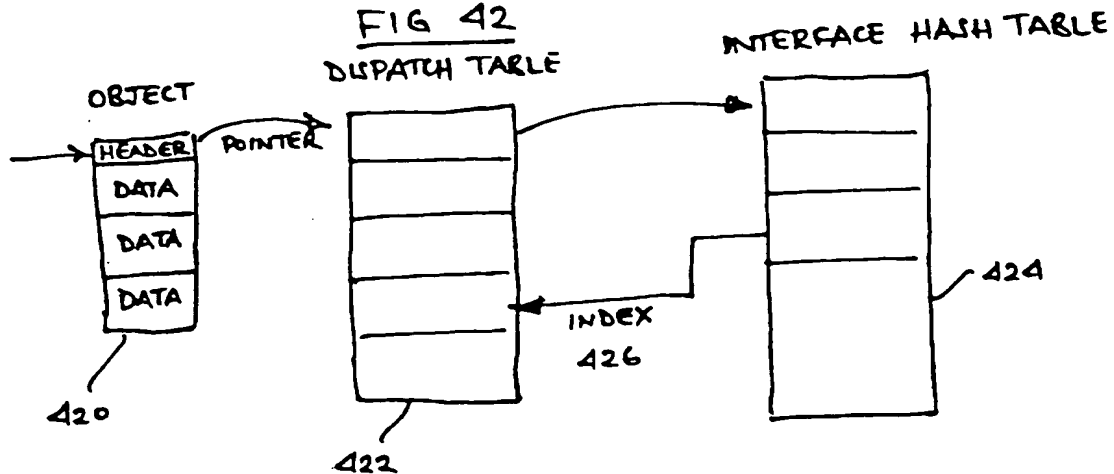
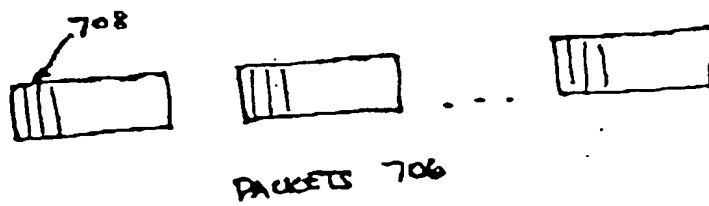
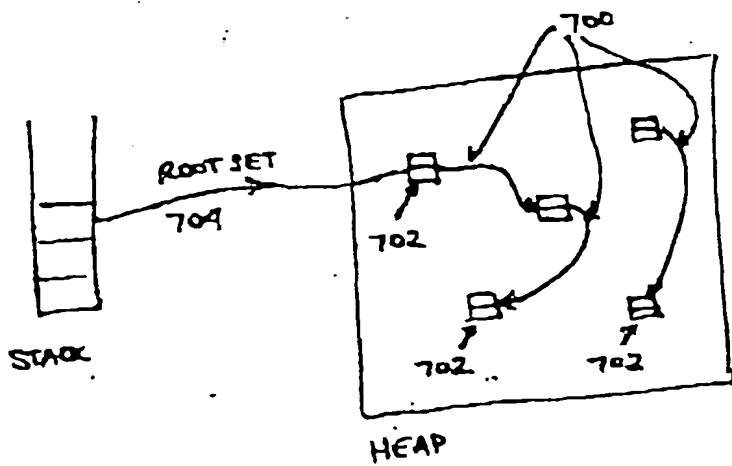


FIG 42
DISPATCH TABLE



THIS PAGE BLANK (USPTO)

Fig 70



THIS PAGE BLANK (USPTO)

FIG 100

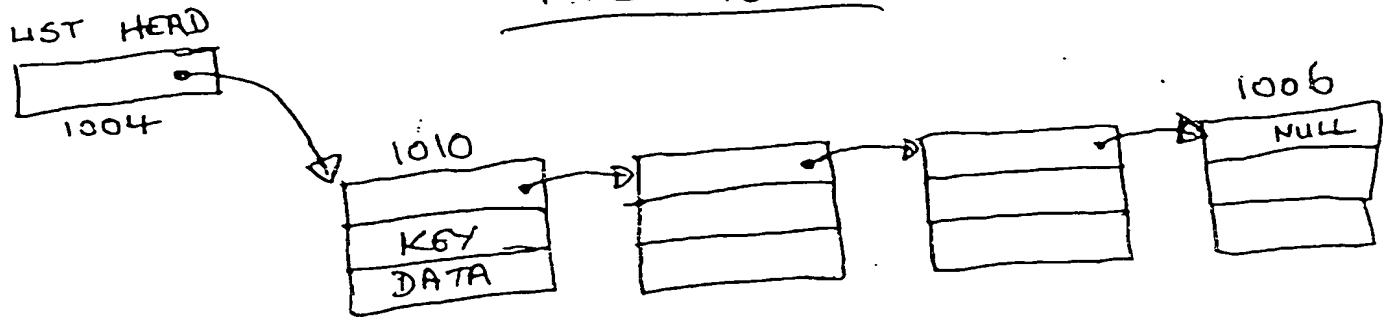


FIG 101

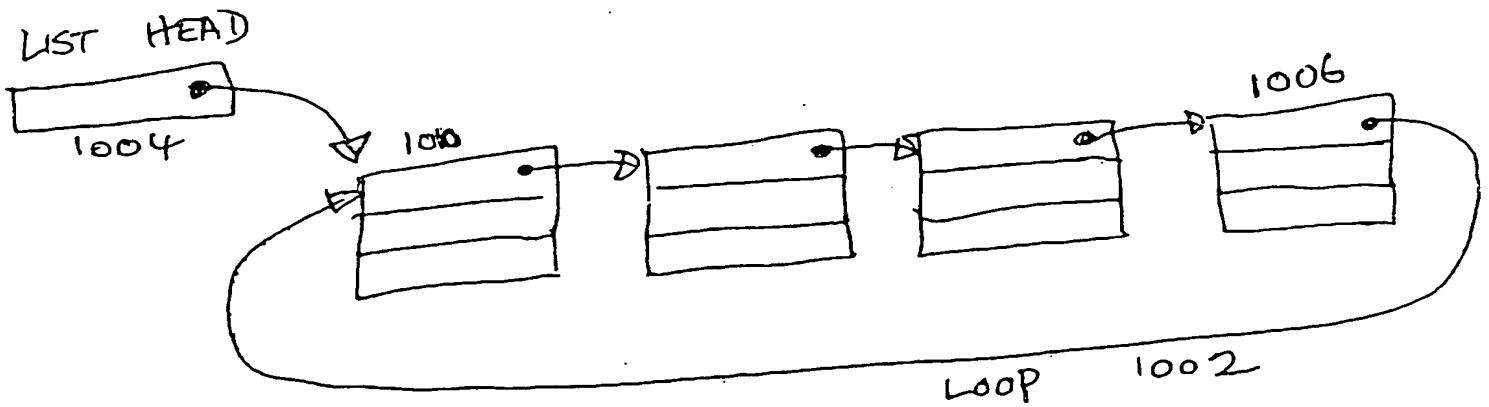
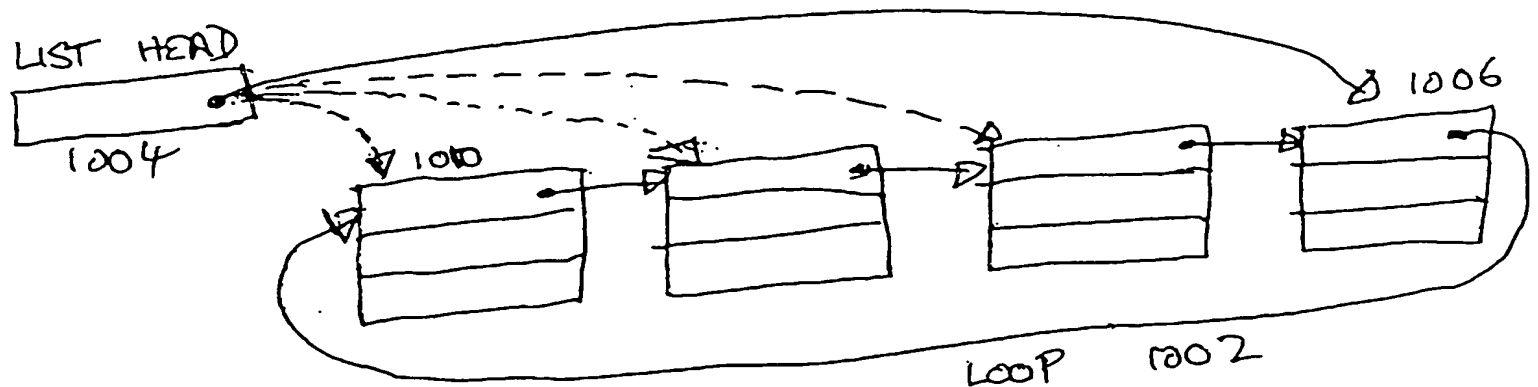
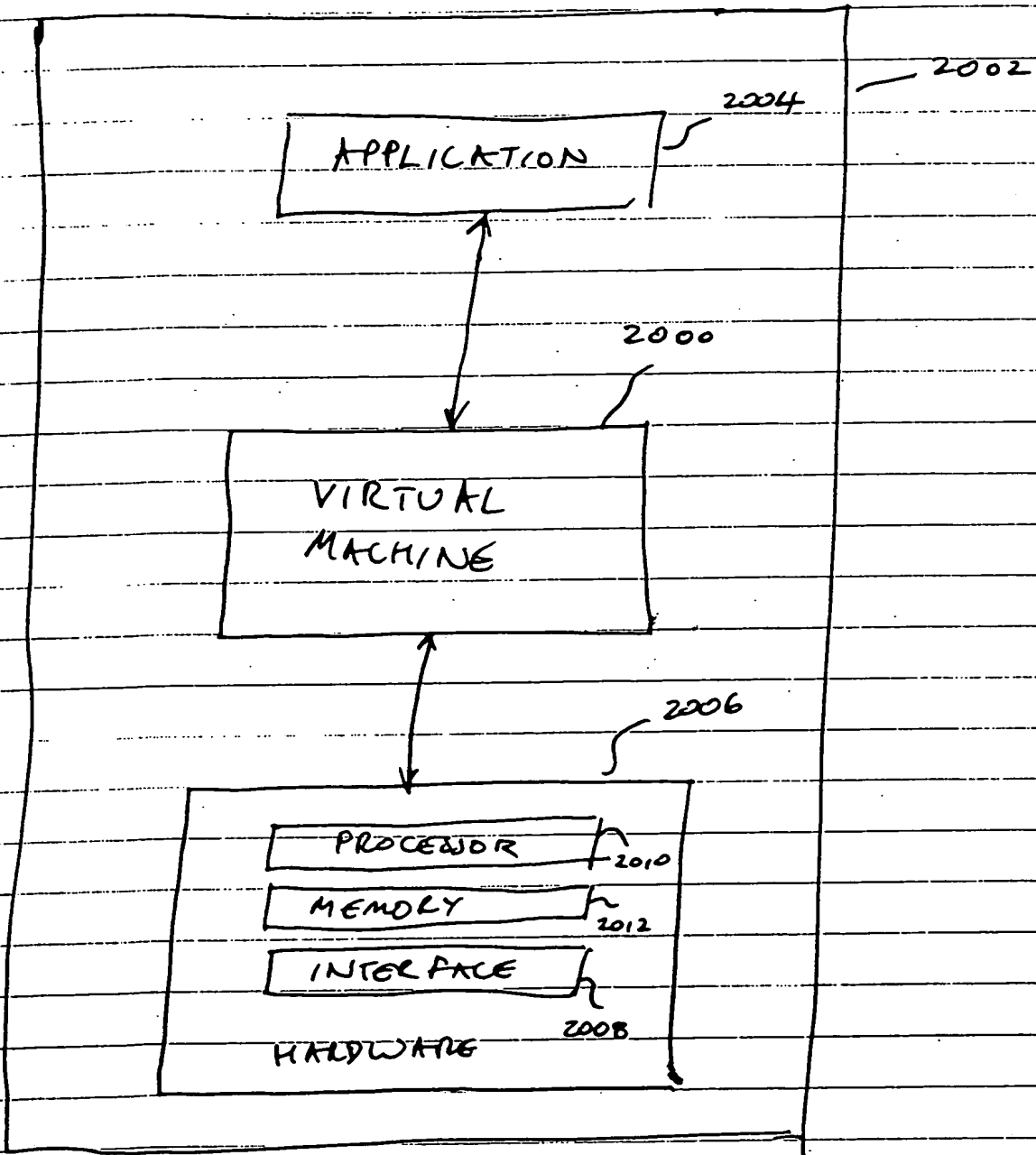


FIG 102



THIS PAGE BLANK (USPTO)

FIG 200



THIS PAGE BLANK (USPTO)

THIS PAGE BLANK (USPTO)